Advanced I/O

- Exceptions
- Binary files
- Sequential/Random access
- Serializing objects
Exceptions

- No matter how good of a programmer you are, you can’t control everything.
  - Other users
  - Available memory
  - File allocation
- How do we write a safe method that behaves *sanely* even in the possibility of failure for reasons beyond your control?
  - Exceptions: tell the calling code “Something bad happened. I failed.”
  - Any method that take a risk (could fail) should declare that risk when you take it so that the failure can be dealt with if it occurs
  - A try/catch block tells the compiler that you *know* you are calling something that might fail at run-time and what to do if it *does* fail.
What is an Exception?

- In Java, an exception is an object
  - You can make your own!
- Exceptions are generated (thrown) by methods that perform risky behavior
- Exceptions that are of type RuntimeException (or its subclasses) are unchecked by the compiler
- The compiler forces code to deal with all other types of exceptions
Generating an Exception

```java
public class RobotException extends Exception {
    public RobotException { super("Error: Bad robot!"}; }
} // end class RobotException

public class RobotOnFireException extends RobotException {
} // end class RobotOnFireException

public class CrazedRobotException extends RobotException {
} // end class CrazedRobotException

public void useRobot (Order order) throws RobotException {
    boolean taskPerformed = giveRobotOrder(Order order);
    if (!taskPerformed()) {
        if (isRobotOnFire()) {
            throw new RobotOnFireException();
        }
        if (RobotIQ > 200) { throw new CrazedRobotException(); }
    }
} // end useRobot

● Exceptions declare a problem, they don’t handle/fix it! That’s for the person who called the code to deal with!
```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
   String index out of range: 3
   at java.lang.String.charAt(Unknown Source)
   at Coordinate.getStatus (Coordinate: 67)
   at Ship.getLocation (Ship: 15)
   at GameBoard.checkTarget (GameBoard.java: 108)
   at GameBoard.play (GameBoard.java: 54)
   at Main.main (Main.java: 23)
Ducking responsibility: throw

```java
import java.io.File;
import java.util.Scanner;
public class Main {
    public static void main (String[] args) throws Exception {
        String filename = "Data.txt"; // in working directory
        File fileHandle = new File(filename);
        Scanner inputFile = new Scanner (fileHandle);

        String line;
        while ( inputFile.hasNextLine() ) {
            line = inputFile.nextLine();
            System.out.println(line);
        }
        inputFile.close();
    } // end method main
} // end class Main
```
Taking responsibility: Try/Catch

- The compiler needs you to acknowledge that you’ve taken responsibility for handling an exception
- Throwing it up the call stack acknowledges it without doing anything at that level
- Try/Catch allows you to deal with the problem and put it to bed
- If an exception passed through main, JVM will shut down

```java
try {
    // potentially risky code
    robot.useRobot(order);
} catch (RobotOnFireException e) {
    useFireExtinguisher(robot);
} catch (CrazedRobotException e) {
    startVirus(robot);
} catch (Exception e) {
    System.out.println("Robot " + "is out of control.  Run!");
}
```
public static void main (String[] args) {
    boolean validFile;

    do {
        String filename = getFilenameFromUser();
        File fileHandle = new File (filename);
        Scanner inputFile = null;
        validFile = true;
        try {
            inputFile = new Scanner (fileHandle);
        } catch (Exception e) {
            System.out.println("Error opening input file: "+filename);
            validFile = false;
        }
        until (validFile);

        processFile(inputFile);
    } // end method main
Binary files

- Text files
  - All data store in character format (UNICODE, ASCII, etc).
  - Number 4 stored as one character ‘4’
  - Number 10000000 stored as 8 characters ‘1’ ‘0’ ‘0’ … ‘0’
  - Each character is 16-bits in unicode

- Binary files
  - All data stored in “raw binary” format (as in computer memory)
  - Number 4 is stored as one 32-bit value (x0000 0004)
  - Number 10000000 is stored as one 32-bit value
  - Far more efficient (few conversions, usually less space)
  - Some things have no text equivalent (images, etc)
Writing to a binary file

import java.io.FileOutputStream;
import java.io.DataOutputStream;
...
public void storeToFile(ArrayList<Integer> numberList) {
    try {
        FileOutputStream fs = new FileOutputStream(filename);
        DataOutputStream output = new DataOutputStream(fs);
        for (int number : numberList) {
            output.writeInt (number);
        }
        output.close ();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
} // end method storeToFile
import java.io.FileInputStream;
import java.io.DataInputStream;
...
public ArrayList<Integer> readFromFile() {
    ArrayList<Integer> numberList = new ArrayList<Integer>();
    boolean endOfFile = false;
    try {
        FileInputStream fs = new FileInputStream(filename);
        DataInputStream input = new DataInputStream(fs);
        while (!endOfFile) {
            numberList.add( input.readInt() );
        }
    } catch (Exception ex) {
        if (ex instanceof EOFException) { endOfFile = true; }
        else {ex.printStackTrace();}
    }
    input.close();
} // end method readFromFile

Value written to

Breaks into bytes chained to

DataOutputStream

Reads bytes to file chained to

FileOutputStream

File
Random Access Files

- Sequential access
  - Opening a file sets the File Pointer to Byte 0.
  - Reading/Writing advances the File Pointer the correct number of Bytes.
- Random access
  - Opening a file sets the File Pointer to Byte 0.
  - Reading/Writing advances the File Pointer the correct number of Bytes
  - The File Pointer can be changed without reading
- Class RandomAccessFile
  - Opens files for read, read/write
  - Mode “r” Read
  - Mode “w” Write
Reading/Writing from a RandomAccessFile

```java
import java.io.RandomAccessFile;
public class Main {

    public static void main (String[] args) throws Exception {
        final int SIZE_OF_INT_IN_BYTES = 4;
        RandomAccessFile file = new RandomAccessFile("Data.dat","rw");
        file.writeInt(7000);
        file.writeInt(102);
        file.writeInt(123456);
        file.seek(1 * SIZE_OF_INT_IN_BYTES);
        System.out.println(file.readInt());
        file.close();
    } // end method main

} // end class Main
```
Serializing objects

• How can we store/load an object between runs?
• An instance of an object is really just:
  – The specific values of the instance variables (its state)
  – Stored in a specific order somewhere in the heap
  – With a reference that tells you where to find the structure
• If the instance variables are all primitive values, this is easy
• If the instance variables contain object references (aggregation), then the specific values of the aggregate objects are also part of the object’s effective state.
• Essentially, to store a complex object, we need to flatten it out and all the objects it knows about (and so on) to a “flat” savable representation.
• This process is known as serialization
• In Java, classes which implement java.io.Serializable can be serialized
import java.io.Serializable;

public class Card implements Serializable {
    String suit;
    String rank;

    public Card (String suit, String rank) {
        this.suit = suit;
        this.rank = rank;
    } // end constructor

    public String toString() {
        return (suit + " of " + rank + "s");
    }

} // end class Card
Example: Deck of Cards

```java
// class header and state
public class Deck implements Serializable {

    // class methods
    public Deck () {
        static final String[] suitList = {
            "Club", "Diamond", "Heart", "Spade"};

        static final String[] rankList = {
            "Ace", "Two", "Three", "Four",
            "Five", "Six", "Seven", "Eight",
            "Nine", "Ten", "Jack", "Queen",
            "King"};

        ArrayList<Card> cardList =
            new ArrayList<Card>();

        for (String suit : suitList) {
            for (String rank : rankList) {
                cardList.add(new Card(rank, suit));
            }
        }

    }

    public void showDeck () {
        for (Card card : cardList) {
            System.out.println(card);
        }
    }

    public static void main (String[] args) {
        Deck deck = new Deck();
        deck.storeToFile("deckFile.dat");
        deck = deck.getFromFile("deckFile.dat");
        deck.showDeck();
    }

    public void storeToFile(String filename) {
        // code to store deck
    }

    public Deck getFromFile(String filename) {
        // code to get deck
    }
}
```
import java.io.Serializable;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
...

public void storeToFile(String filename) {
    try {
        FileOutputStream fs = new FileOutputStream(filename);
        ObjectOutputStream os = new ObjectOutputStream(fs);
        os.writeObject (this);
        os.close ();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
} // end method storeToFile
import java.io.Serializable;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
...

public Deck getFromFile(String filename) {
    Deck storedDeck = null;
    try {
        FileInputStream fs = new FileInputStream(filename);
        ObjectInputStream is = new ObjectInputStream(fs);
        storedDeck = (Deck) is.readObject();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    return storedDeck;
} // end method getFromFile